VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Úprava projektu BP Studio

Extension of BP Studio

2015 Ondřej Horáček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Horáček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Úprava projektu BP Studio
Extension of BP Studio

Zásady pro vypracování:

Cílem práce je provést následující změny v projektu BP Studio (JAVA):
1. Využití generických kolekcí místo starých negenerických kolekcí.
2. Vylepšení grafické reprezentace výsledku simulací (zavedení grafů - Gantt chart, histogram, ...).
3. Rozšíření grafického rozhraní a přidělení dalších atributů uzlům grafu.
4. Vytvoření UML dokumentace stávajícího stavu projektu.
5. Implementovat funkci zvětšení a zmenšení, také do zobrazení simulačních diagramů.
6. Rozšíření popisů ikon a hran v jednotlivých diagramech.
7. Možnost nastavení volby způsobu výběru scénářů.
8. Monitorování průběhu simulací.

Práce bude obsahovat:
1. Popis současného stavu projektu.
2. Přehled dostupných knihoven pro tvorbu grafů a jejich zhodnocení.
3. Analýzu požadovaných změn.
4. Implementaci výše uvedených změn.
5. Dokumentaci a popis softwaru a provedených změn.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] HOEBER, Mark, et al. Java 6 Výukový kurz. Brno : COMPUTER PRESS, 2007. 534 s. EAN: 9788025115756

Dále podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 7. května 2015 ..... Horáček

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2015 ..... Horáček

Rád bych na tomto místě poděkoval vedoucímu mé bakalářské práce Ing. Davidu Ježkovi, Ph.D., za veškeré cenné rady, pomoc s orientací v projektu a za trpělivost při vytváření této práce.

Abstrakt

Cílem této bakalářské práce je seznámení se s programem BP Studio, který je implementován v jazyce Java, a provedení úprav na tomto programu. Nejprve je v práci nezbytná úprava již existujícího zdrojového kódu a následně jsou rozšířeny i možnosti programu. Čtenář je v rámci této práce obeznámen s tím, co všechno obnáší takové úpravy prováděné na rozsáhlejším projektu.

Klíčová slova: bakalářská práce, BP Studio, Java, grafická knihovna

Abstract

Objective of this Bachelor Thesis is to become familiar with the BP Studio program, which is implemented in Java, and performing required changes into it. First, there is need to perform necessary adjustment of existing source code, and then the program options are extended. The reader is familiarised with what is it all about performing such modifications on larger project.

Keywords: Bachelor Thesis, BP Studio, Java, graphic library

Seznam použitých zkratek a symbolů

UML	–	Unified Modeling Language
API	–	Application Programming Interface
JDK	–	Java Development Kit
PDF	–	Portable Document Format
SVG	–	Scalable Vector Graphics

Obsah

1 Úvod **4**

2 BP Studio **5**
2.1 Modelování 5
2.2 Simulace 8

3 Analýza změn **11**

4 Implementace změn **13**
4.1 Přepis kódu na využití generických kolekcí 13
4.2 Implementace funkce zvětšení a zmenšení 16
4.3 Zobrazení aktuálního času simulace 16
4.4 Možnost zakázat paralelní vykonávání aktivity 17
4.5 Možnost skrytí aktivit ve výsledcích simulace 17
4.6 Implementace možnosti volby scénáře 18
4.7 Úprava vzhledu volby násobnosti 19
4.8 Úpravy a přidávání tabulek s výsledky simulace 20
4.9 Knihovny pro tvorbu grafů 22
4.10 Implementace grafů 23

5 Závěr **29**

6 Reference **30**

Přílohy **30**

A CD s aplikací **31**

Seznam obrázků

1 UML diagram zobrazující strukturu modelu 7
2 UML diagram zobrazující strukturu tříd pro simulaci 10
3 Paralelní vykonávání aktivity 17
4 Aktivity skryté ve výsledcích 18
5 Možnost volby scénáře 19
6 Změna vzhledu volby násobnosti 19

Seznam výpisů zdrojového kódu

1 Ukázka kódu třídy ExtendedAbstractTableModel 14
2 Ukázka kódu třídy RecordStatisticTableModel 14
3 Ukázka kódu vnořené třídy reprezentující datový model tabulky 15
4 Metoda pro přepočet rozměrů na základě poměru zoomu 16
5 Ukázka použití GridBagLayout a GridBagConstraints 20
6 Ukázka vytvoření JTableButtonRenderer třídy pro zobrazení tlačítka v tabulce 21
7 Ukázka vytvoření instancí rendererů a jejich využití 21
8 Ukázka třídy pro uchování počtu tokenů v uzlech v určitém čase simulace 22
9 Získání seznamů dob začátků a ukončení aktivit 24
10 Získání hodnot pro vykreslení grafu 24
11 Vytvoření a naplnění datasetu 25
12 Získání minimální a maximální doby trvání aktivity 26
13 Získání hraničních hodnot intervalů 26
14 Vytvoření a naplnění bloků histogramu 27
15 Naplnění datasetu 28
16 Metoda pro vytváření datasetu u grafu zobrazujícího pohyb tokenů v uzlu 28

1 Úvod

Modelování podnikových procesů je v současné době důležitou součástí při vytváření podnikových procesů. Umožňuje získat přehled o skupině aktivit a úloh, které jsou potřeba pro vykonání určité služby nebo vyrobení výrobku. Je také žádoucí, aby tyto procesy byly co nejefektivnější, tudíž je možné procesy díky těmto modelům optimalizovat. A přesně k tomu slouží program BP Studio.

Textová část práce je rozdělena na 3 hlavní kapitoly. První kapitola popisuje blíže účel programu BP Studio a jeho strukturu. Zabývá se popisem tříd sloužících pro vytváření modelů podnikových procesů v aplikaci, tříd sloužících pro generování simulovaných modelů a tříd využívaných přímo při simulaci. Dále je zde popsán druh simulace využívaný pro simulování podnikových procesů a postup, jak probíhá tato simulace v programu BP Studio.

Druhá kapitola se zabývá analýzou žádoucích změn v aplikaci. Jsou zde popsány změny prováděné v aplikaci, důvody vedoucí k provádění těchto změn, případně důvody vedoucí k implementaci nových možností do programu.

Třetí kapitola už je následně věnována přímo implementaci změn popsaných ve druhé kapitole. Je zde popsán detailní postup implementace jednotlivých změn. Také je v této kapitole uveden přehled knihoven pro vytváření grafů v jazyce Java a postupy při vytváření grafů za využití zvolené knihovny.

2 BP Studio

BP Studio je nástroj umožňující vytváření modelů podnikových procesů, které mohou být poté analyzovány, případně vylepšovány, aby se zvýšila kvalita procesu, snížila se doba cyklu procesu nebo např. snížila cena nákladů pro určitý proces. BP Studio nabízí možnost vytváření funkčních, objektových nebo koordinačních modelů. Koordinační modely je také možné následně odsimulovat a získat tak informace o tom, jak by mohl proces v praxi vypadat, zda se v něm nenachází nějaký slepý bod, jak dlouho by jednotlivé aktivity a proces celkově trval, které aktivní a pasivní objekty jsou nezbytné pro zahájení aktivity, jaké náklady by potřebovala která aktivita atd.[1]

Program by se dal tedy rozdělit na 2 hlavní části. První z nich je modelování a analýza procesu. Tou druhou částí je poté v případě koordinačního modelu možnost simulace tohoto procesu.

2.1 Modelování

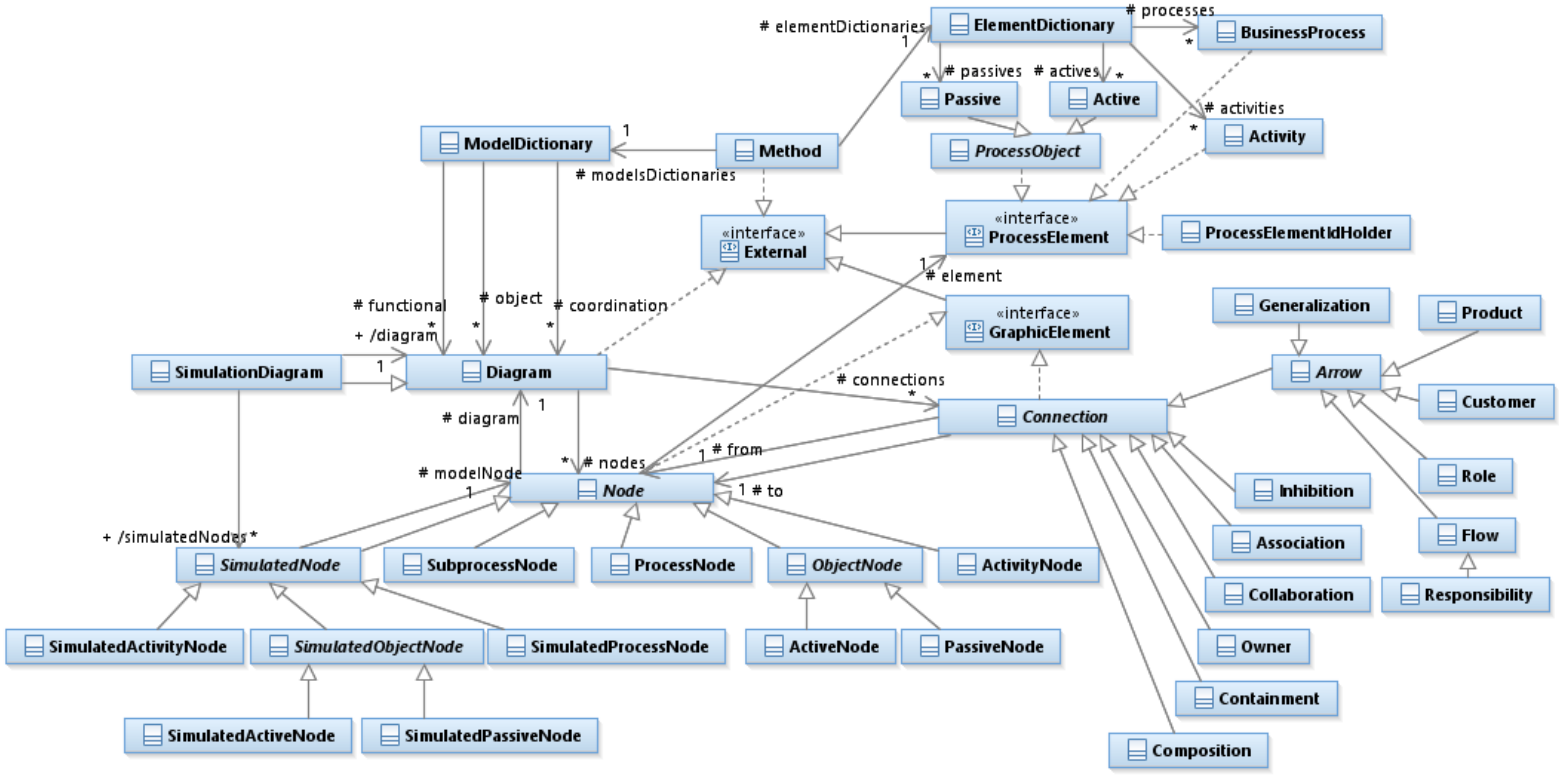
Hlavní třídou pro vytváření modelů je třída Method. Ta se stará o přidávání diagramů a elementů, také o získávání jejich hodnot a případně o slučování více instancí třídy Method. (struktura tříd určených pro modelování viz. obrázek 1)

Pro práci s elementy uvnitř třídy Method slouží instance třídy ElementDictionary. Zde se ukládají kolekce všech podnikových procesů, aktivních prvků, pasivních prvků a aktivit (třídy BusinessProcess, Active, Passive a Activity). Všechny tyto 4 třídy implementují rozhraní ProcessElement, což je základní rozhraní pro všechny elementy procesu. Třídy Active a Passive toto rozhraní neimplementují přímo, ale dědí ze třídy ProcessObject, která rozšiřuje možnosti rozhraní ProcessElement o další vlastnosti pro aktivní a pasivní objekty.

Pro práci s diagramy ve třídě Method slouží instance třídy ModelDictionary. Zde jsou pro změnu uloženy kolekce všech vytvořených funkčních, objektových a koordinačních diagramů (třída Diagram). Třída Diagram je základní třídou pro všechny diagramy. Uchovává v sobě kolekci uzlů (třída Node) a propojení uzlů (třída Connection), které jsou vykresleny na tomto diagramu. Jak třída Node, tak i třída Connection implementují rozhraní GraphicElement, což je základní rozhraní pro všechny objekty, které lze vykreslit.

Třída Node je základní třídou pro všechny vykreslované uzly diagramu. Uchovává si v sobě odkaz na diagram, na kterém je vykreslen, a také odkaz na instanci třídy ProcessElement, tedy zobrazovaného objektu. Třída Node je dále rozšiřována pro jednotlivé vykreslované elementy, kde každý z těchto potomků implementuje metody pro vykreslení specifického vzhledu uzlu. Třída SubprocessNode slouží pro grafické zobrazení instance třídy BusinessProcess v kontextu koordinačního diagramu, třída ProcessNode slouží pro zobrazení instancí třídy BusinessProcess i v ostatních diagramech, třída ActivityNode reprezentuje grafické zobrazení instancí třídy Activity a třída ObjectNode rozšiřuje možnosti třídy Node o některé vlastnosti specifické pro aktivní a pasivní uzly. Třídu ObjectNode poté rozšiřuje třída ActiveNode pro vykreslení aktivních objektů (instance třídy Active) a třída PassiveNode pro vykreslení pasivních objektů (instance třídy Passive).

Třída Connection je základní třídou pro všechny vykreslované propojení uzlů v diagramu. Každá instance si uchovává odkaz na dvě instance třídy Node, do kterých ukládá uzly, mezi kterými je toto propojení nastaveno. Stejně jako tomu bylo i u třídy Node, tak i třída Connection je dále rozšiřována pro zobrazení specifického vzhledu propojení mezi jednotlivými uzly. Pro zobrazení vztahů mezi uzly ve funkčním diagramu slouží třídy Collaboration reprezentující vztah spolupráce, Owner reprezentující vztah vlastníka uzlu, Containment reprezentující vztah obsažení (proces obsahuje nějaký podproces), Product reprezentující vztah produktu (jaké aktivní a pasivní objekty jsou produkty procesu) a Customer reprezentující vztah zákazníka (jaké aktivní a pasivní objekty jsou potřeba pro zahájení procesu). Vztahy mezi uzly v objektovém diagramu jsou reprezentovány třídami Association pro zobrazení asociace mezi objekty, Composition pro zobrazení vztahu kompozice, Generalization pro zobrazení vztahu zobecnění a Role pro zobrazení vztahu role. Nakonec pro vykreslení vztahů mezi uzly v koordinačním diagramu slouží třídy Flow zobrazující vztah toku (jaké objekty jsou vstupy do aktivity, případně její výstupy), Responsibility rozšiřující třídu Flow a zobrazující vztah zodpovědnosti (speciální případ vztahu toku určující, jaký aktivní objekt je zodpovědný za aktivitu) a třída Inhibition zobrazující vztah inhibice (jaký objekt nesmí být přítomen pro vykonání aktivity). Zatímco třídy Collaboration, Owner, Containment, Association, Composition a Inhibition rozšiřují přímo třídu Connection, třídy Product, Customer, Generalization, Role a Flow dědí ze třídy Arrow, která rozšiřuje třídu Connection o metody pro vykreslení šipky a slouží jako základní třída pro všechny tyto šipkové vztahy.



Obrázek 1: UML diagram zobrazující strukturu modelu

2.2 Simulace

2.2.1 Diskrétní simulace

Pro simulaci procesů se v programu BP Studio využívá diskrétní (nespojitá) simulace. Oproti spojité simulaci, kde je čas rozdělen na určité intervaly a stav systému se mění podle množiny činností, které se odehrávají v daném intervalu, je u diskrétní simulace průběh simulace rozdělen na posloupnost událostí v čase. Každá z těchto událostí se vyskytuje v určitém čase a značí změnu stavu systému. Protože je každá událost zaregistrována, tak mezi po sobě jdoucími událostmi se nepředpokládá žádná změna v systému a tudíž lze v simulaci poskočit v čase hned z jedné události na následující. Díky tomu, že při diskrétní simulaci není nutné simulovat každý časový okamžik tak, jako u spojité simulace, může tato simulace běžet mnohem rychleji.[2]

Mezi komponenty diskrétní simulace patří:

- Stav - množina proměnných zachycující charakteristické vlastnosti systému určené k prostudování
- Hodiny - simulace musí sledovat aktuální čas simulace v jednotkách vhodných pro daný systém. Na rozdíl od simulace v reálném čase, čas při diskrétní simulaci se mění skokově
- Seznam událostí - simulace musí mít alespoň jeden seznam událostí, které se mají vykonat. Každá událost má daný čas, ve kterém se má vykonat, a typ určující, jaký kód má být použit pro simulaci této události
- Generátory náhodných čísel - simulace potřebuje generovat náhodné hodnoty různých druhů v závislosti na modelu systému
- Statistiky - simulace si typicky uchovává výsledky simulace systému pro jejich analýzu
- Koncové podmínky - teoreticky by diskrétní simulace mohly běžet nepřetržitě, protože události jsou samospoušťecí. Proto je zapotřebí nastavit i podmínku, kdy simulace skončí.

2.2.2 Průběh simulace (struktura tříd viz. obrázek 2)

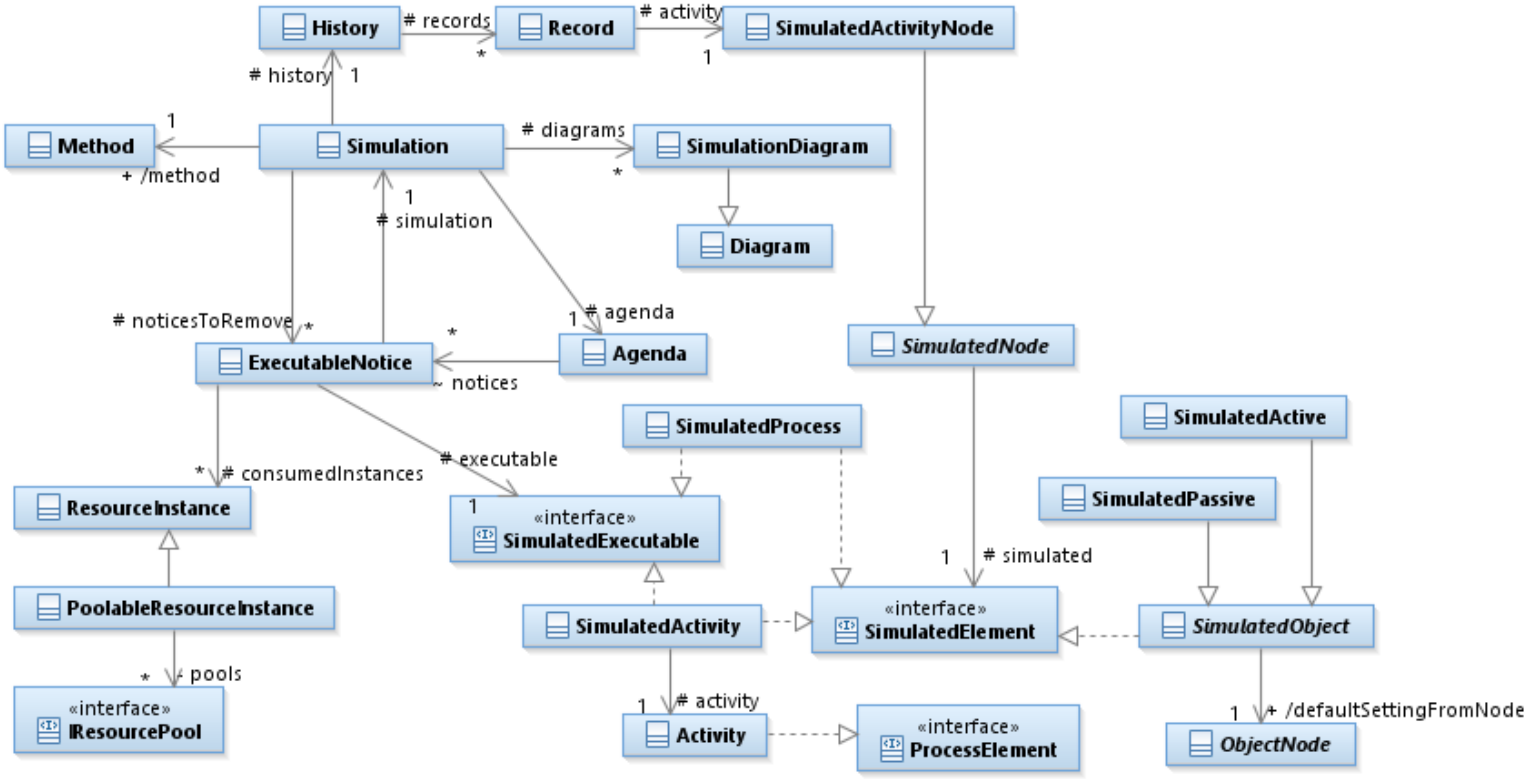
Při přepnutí do režimu simulace se projdou všechny diagramy ze třídy Method a pro každý koordinační diagram se vytvoří instance třídy SimulationDiagram, která rozšiřuje možnosti třídy Diagram o nezbytné metody pro simulaci a ukládá v sobě odkaz na instanci třídy Diagram, ze které je vytvořen. Instance třídy SimulationDiagram je následně uložena do kolekce uvnitř třídy Simulation. Dále se projdou všechny uzly diagramu a vytvoří se z nich instance třídy SimulatedNode. Tato třída obsahuje referenci na instanci třídy Node, ze které je vytvořen, a taky obsahuje odkaz na SimulatedElement.

Třída SimulatedNode je základní třídou pro všechny simulovatelné uzly a je následně rozšiřována potomky této třídy, kde každý z potomků rozšiřuje možnosti třídy o vykreslení daného typu uzlu. Mezi třídy přidávající možnosti grafického zobrazení uzlu patří třídy SimulatedActivityNode pro reprezentaci simulované aktivity, SimulatedProcessNode pro reprezentaci simulovaného procesu a třída SimulatedObjectNode, která je dále rozšířena třídou SimulatedActiveNode pro reprezentaci simulovaného aktivního objektu a třídou SimulatedPassiveNode pro reprezentaci simulovaného pasivního objektu.

Rozhraní SimulatedElement je základní rozhraní pro všechny simulované elementy a je implementováno třídou SimulatedActivity, která představuje simulovanou aktivitu a obsahuje odkaz na příslušnou instanci třídy Activity. Dále rozhraní SimulatedElement implementuje třída SimulatedProcess reprezentující simulovaný proces a také třída SimulatedObject, která představuje aktivní a pasivní objekty v průběhu simulace. Třída SimulatedObject je dále rozdělena na třídy SimulatedActive a SimulatedPassive pro aktivní a pasivní objekty.

Třídy SimulatedActivity a SimulatedProcess implementují kromě rozhraní SimulatedElement i rozhraní SimulatedExecutable, což je rozhraní reprezentující všechny proveditelné elementy v simulaci.

O hlavní průběh simulace se stará třída Simulation. Zde se po zahájení simulace pro každou událost - krok simulace nejprve projdou všechny simulovatelné aktivity a zjistí se, zda jde některou z nich zahájit. Pokud ano, tak se vloží záznam do třídy Agenda, která se stará o plánování simulace procesu. Třída Agenda si uchovává kolekci instancí třídy ExecutableNotice, která je využita při plánování agendy a uchovává referenci na příslušný element implementující rozhraní SimulatedExecutable, referenci na danou simulaci a čas, kdy dojde k další události daného objektu. Při vložení záznamu do třídy Agenda se uvnitř třídy vytvoří nová instance této třídy ExecutableNotice a je podle času zařazena na správnou pozici v kolekci, aby byly tyto instance chronologicky seřazeny. Následně se nastaví čas uvnitř Agendy na nadcházející událost a zjistí se, které aktivity se v tomto čase mohou dokončit a příslušné instance třídy ExecutableNotice jsou poté odebrány z Agendy. Nakonec se projdou všechny uzly v diagramech a aktualizují se. V každém kroku simulace se také zaznamenávají do historie (třída History) nové záznamy (instance třídy Record) obsahující informace o simulovaných aktivitách ve všech diagramech, které jsou načteny v simulaci. Tyto záznamy jsou po skončení simulace využity pro zobrazení výsledků simulace.



Obrázek 2: UML diagram zobrazující strukturu tříd pro simulaci

3 Analýza změn

Při prvním pohledu na kód původní verze BP Studia mohlo být pro kteroukoliv osobu dosti obtížné a matoucí zorientovat se v tomto kódu. Také při manipulaci a úpravách v kódu bylo snadné se zamotat. Velký podíl na tom mělo i využití třídy Vector bez specifikování typu vektoru. Kvůli tomu bylo často potřeba hledat v kódu, co se do dané proměnné typu Vector ukládá za hodnoty. O to více matoucí byly vektory uchovávající v sobě hodnoty dalších vektorů nebo dokonce i vektory uchovávající další vektory vektorů. Z tohoto důvodu bylo potřeba co nejvíce z těchto vektorů přepsat za využití generických kolekcí. Nakonec jsem se tedy rozhodl přepsat vektory na generické ArrayListy.

Dalším krokem při úpravě projektu bylo doplnění možnosti přiblížení a oddálení plátna pro lepší přehled v diagramu, obzvláště u složitějších diagramů obsahujících více prvků. Tato možnost již byla implementována u modelování diagramu, ovšem při přepnutí do módu simulace již tato možnost chyběla. Proto bylo požadováno její doplnění.

Dále při spuštění simulace se po jejím dokončení zobrazí popisek oznamující, že simulace byla dokončena. Ovšem při spouštění složitějších procesů může tato simulace trvat i několik sekund nebo dokonce i déle. Během této doby uživatel neví, zda simulace ještě běží nebo co se děje. Proto bylo potřebné nějakým způsobem monitorovat i průběh simulace.

Ne vždy se může při vykonávání procesu každá aktivita vykonávat vícekrát současně. Může nastat případ, kdy daná aktivita smí být prováděna v určitou chvíli pouze jednou. Ovšem tohoto nebylo možné docílit při simulaci v původní verzi projektu a tak se vždy, při dostatečném počtu vstupujících instancí do aktivity, začala vykonávat i vícekrát paralelně. Aby bylo možné zvolit, zda určitá aktivita smí nebo nesmí běžet vícekrát v daném čase, bylo nezbytné tuto možnost také naimplementovat.

Při vytváření procesu mohou být přidány v BP Studiu aktivity, které nejsou úplně důležité (např. nějaký log) a bylo by žádoucí, aby kvůli přehlednosti nebyly tyto aktivity zobrazovány ve výsledné tabulce průběhů aktivit. Za cílem umožnění této volby byla implementována možnost výběru u aktivit, zda se mají nebo nemají zobrazovat ve výsledcích.

Také se u procesů mohou často objevit aktivity, které nejsou úplně přímočaré, tedy neřídí se vždy jen jedním scénářem, ale mohou mít více scénářů, kde každý z nich může mít jiné náklady, trvat rozlišnou dobu, vést k různým následujícím aktivitám apod. Vezmeme si například jako aktivitu nějakou revizi. Chceme, aby tato revize při simulaci z 90 procent proběhla úspěšně, ale z 10 procent, aby revize našla chybu. Při simulaci se poté podle procent vykonání daných scénářů vykoná jeden z nich. Jenže co když chceme odsimulovat přímo jen cestu, která se vykoná v případě vykonání scénáře s 10 procentní šancí? V původní verzi bylo nutné spouštět simulaci opakovaně, dokud neproběhl námi požadovaný scénář, což není zrovna ideální řešení. Z tohoto důvodu jsem naimplementoval i možnost volby scénáře při simulaci aktivity.

U propojení uzlů v diagramu lze nastavit kromě jiných nastavení i počet instancí, které jsou potřeba z daného aktivního či pasivního prvku pro vykonání aktivity, popř. kolik instancí vznikne po vykonání aktivity. Tento počet lze nastavit přímo na určitý počet (násobnost) nebo na rozsah od minimálního do maximálního počtu nebo na náhodný

počet v určitém rozsahu. V původní verzi BP Studia je toto řešeno tak, že jako výchozí se bere přímo hodnota násobnosti, popřípadě pokud je vyplněn rozsah minima a maxima, vezme se tato hodnota nebo pokud je nastavena náhodná hodnota v určitém rozsahu, použije se tato. V diagramu jsou poté u propojení uzlů zobrazeny všechny nastavené hodnoty a rozsahy. Ovšem tato funkcionalita nemusí být úplně jednoznačná nebo přehledná pro uživatele, a proto bylo potřeba upravit i vzhled této volby.

4 Implementace změn

4.1 Přepis kódu na využití generických kolekcí

I když se může na první pohled zdát, že přepsání vektorů na generické arrayListy je jednoduché, nemusí to být vždy pravda. Nejde jen o změnu tříd z Vector na ArrayList, jak to může vypadat. Je třeba také nahlédnout hlouběji do kódu a zjistit, jaké hodnoty se do tohoto vektoru ukládají. Často jsou tyto vektory naplněny hodnotami, které jsou předány nějaké metodě v parametru. Tudíž je potřeba se občas zanořit ještě hlouběji do kódu a to jen pro zjištění hodnot, které se do vektoru ukládají. Tyto vektory se následně procházely v cyklech za využití třídy Enumeration a při přepsání vektorů na arrayListy nebylo možné tyto cykly využít, tudíž musely být taky přepsány za využití jiného druhu cyklu.

Dalším z problémů, které se vyskytly po přepsání kódu, byly metody, které měly jako návratový typ funkce např. `ArrayList<? extends ProcessElement>` a podle typu uvedeného v parametru vracely ArrayList instancí třídy Activity, BusinessProcess, Active nebo Passive. Občas byly v kódu tyto metody volány za účelem získání přímo specifické kolekce, která byla uložena původně do proměnné typu Vector a v cyklu se jednotlivé hodnoty z tohoto vektoru přetypovaly na instanci specifické třídy, se kterou se dále pracovalo. Ovšem po přepsání těchto vektorů na arraylist stejného typu jako vracela metoda již nebylo možné takto jednoduše projít kolekci a přetypovat hodnoty na instance této specifické třídy, protože mohla tato kolekce teoreticky obsahovat i instance jiné třídy dědící ze třídy ProcessElement. Bylo tedy nutné získat z dané metody přímo kolekci instancí požadované třídy. Proto jsem tento problém nakonec vyřešil tím způsobem, že jsem původní metodu rozepsal na více metod, kde každá vracela přímo kolekci instancí určité třídy.

Nejednou se v kódu také objevily vektory, do nichž se ukládaly různé hodnoty reprezentující vlastnosti, služby nebo scénáře a tyto vektory byly následně uloženy do dalších vektorů. Pro tyto vektory již byly vytvořeny třídy nahrazující dané vnořené vektory (Attribute, Service, Scenario), ovšem nebyly využity. Proto bylo nezbytné vnořené vektory nahradit vytvořením instancí tříd daného typu. Jedním z míst kódu, kde se takovéhle vektory vektorů používaly, bylo při vytváření tabulek pro vkládání atributů a služeb aktivním uzlům, popř. pasivním uzlům a pro vkládání nebo editaci scénářů aktivit. Pro vykreslení těchto tabulek se využívalo komponenty JTable, kde jako TableModel se vkládala instance třídy DefaultTableModel vytvořená z vektoru dat pro tabulku a vektoru názvů hlaviček sloupců vytvářené tabulky. Při přepisu kódu na využití generických kolekcí jsem tento DefaultTableModel nahradil jedním z potomků třídy ExtendedAbstractTableModel(ukázka kódu 1). Tato abstraktní třída rozšiřuje možnosti třídy AbstractTableModel, která je součástí Java Swing knihovny a každý z potomků poté přizpůsobuje vzhled tabulky, implementuje metody pro manipulaci s tabulkou a rovněž uchovává kolekci dat, se kterými se v tabulce pracuje.

```
public abstract class ExtendedAbstractTableModel extends AbstractTableModel {
    public abstract void deleteRow(int index);
    public abstract void insertRow(int index);
    public abstract void addRow();
    public abstract String getDescription(int index);
    public abstract void setDescription(int index, String text);
    public abstract void setEditable(boolean b);
    public ArrayList<Attribute> getAttrs (){
        throw new NotImplementedException();
    }
    public ArrayList<Service> getServices(){
        throw new NotImplementedException();
    }
    public ArrayList<Scenario> getScenarios(){
        throw new NotImplementedException();
    }
}
```

Výpis 1: Ukázka kódu třídy ExtendedAbstractTableModel

Ovšem tabulky se nevyskytují v aplikaci pouze u nastavení uzlů v diagramu, nýbrž i ve výsledcích simulace. Jedním z těchto míst byla tabulka zobrazující informace o průběhu jednotlivých aktivit, jako např. název aktivity, scénář aktivity, který proběhl, čas zahájení a ukončení vykonávání aktivity, doba čekání před vykonáváním další aktivity, doba vykonávání aktivity a náklady na vykonání aktivity. I zde byly data plnící tabulku uloženy v nepřehledných vektorech a nezbývalo nic jiného, než je přepsat. Pro manipulaci s daty jsem vytvořil třídu RecordStatisticTableModel(ukázka kódu 2), která rozšiřuje abstraktní třídu UniversalReadOnlyTableModel dědící ze třídy AbstractTableModel. Třída RecordStatisticTableModel obsahuje jednak záznamy zobrazované v tabulce a jednak i informace o jednotlivých sloupcích tabulky.

```
public class RecordStatisticTableModel extends UniversalReadOnlyTableModel {

    private List<Record> rows;
    private TableColumnDescription[] columns = {
        new TableColumnDescription(Public.texts.getString("Activity"), ProcessElement.class,
         " getActivity ","getElement"),
        new TableColumnDescription(Public.texts.getString("Scenario"), String.class, "getScenario"),
        new TableColumnDescription(Public.texts.getString("Start"), Time.class, "getWaitingStart"),
        new TableColumnDescription(Public.texts.getString("Finish"), Time.class, "getRunFinish"),
        new TableColumnDescription(Public.texts.getString("WaitingDuration"), Time.class,
         "getWaitingDuration"),
        new TableColumnDescription(Public.texts.getString("Duration"), Time.class,
         "getRunDuration"),
        new TableColumnDescription(Public.texts.getString("Costs"), Float.class, "getCosts"),
    };

    public RecordStatisticTableModel(List<Record> results){
        rows = results;
    }

    public void setRows(List<Record> rows) {
```

```
        this.rows = rows;
    }

    @Override
    protected TableColumnDescription[] getColumnsDescription() {
        return columns;
    }

    @Override
    protected Object[] getRows() {
        return rows.toArray();
    }

    public List<Record> getRecords(){
        return rows;
    }
}
```

Výpis 2: Ukázka kódu třídy RecordStatisticTableModel

Následně bylo nutné taky upravit kód vnořené třídy reprezentující datový model zobrazované tabulky tak, aby nezbytné informace získával z instance námi vytvořené třídy(ukázka kódu 3). Největší zásah v této vnořené třídě byl uvnitř metody getValueAt pro získání hodnot pro jednotlivé buňky tabulky. Zde se podle hodnoty sloupce a řádku vybere hodnota pro danou buňku tabulky.

```
class ActivityTraceDataModel extends AbstractTableModel {
    public int getColumnCount() {
        return data.getColumnCount();
    }
    public int getRowCount() {
        return data.getRowCount();
    }
    public Object getValueAt(int row, int col) {
        try {
            Object object = null;
            switch(col){
            case 0: object = data.getRecords().get(row).getActivity().getElement(); break;
            case 1: object = data.getRecords().get(row).getScenario(); break;
            case 2: object = data.getRecords().get(row).getWaitingStart(); break;
            case 3: object = data.getRecords().get(row).getRunFinish(); break;
            case 4: object = data.getRecords().get(row).getWaitingDuration(); break;
            case 5: object = data.getRecords().get(row).getRunDuration(); break;
            case 6: object = data.getRecords().get(row).getCosts(); break;
            }
            return object;
        } catch (Exception e) {
            return null;
        }
    }
    public String getColumnName(int column) {
        return data.getColumnName(column);
    }
    public Class getColumnClass(int c) {
```

```
        return data.getColumnClass(c);
    }
    public boolean isCellEditable(int row, int col) {
        return false;
    }
}
```

Výpis 3: Ukázka kódu vnořené třídy reprezentující datový model tabulky

4.2 Implementace funkce zvětšení a zmenšení

K aplikování možnosti přiblížení a oddálení plátna diagramu bylo nejprve nezbytné přidat tlačítka, které by tento zoom měly na starost. Poté jsem musel vytvořit proměnné, které by reprezentovaly poměr zoomu k původnímu rozměru a vytvořeným tlačítkům přiřadit posluchače, ve kterých by se poměr nastavoval. Také se musela zavolat metoda pro překreslení tohoto plátna. Dále bylo potřeba vytvořit metodu, která by tento poměr aplikovala na jednotlivé uzly a propojení uzlů v diagramu (ukázka kódu 4). Tato metoda získá v parametru instanci třídy Graphics, z té vytvoří kopii a tu přetypuje na instanci třídy Graphics2D, která rozšiřuje možnosti třídy Graphics mimo jiné i o možnosti transformace. Na tuto instanci se poté zavolá metoda scale, která pomocí transformační matice přepočte poměry rozměrů. Velikost následně renderovaných objektů je přepočítána podle tohoto měřítka. Poté se nastaví antialiasing a každý uzel a připojení uzlů v diagramu se vykreslí za využití této přepočtené Graphics. Tímto sice zoom funguje, ale při aktivitách myši (stisk tlačítka, potáhnutí myší atd.) a taky při zobrazení menu při stisku pravého tlačítka myši se stále využívaly původní souřadnice myši bez aplikování zoomu. Proto byla veškerá tato aktivita zkreslená. Toto bylo vyřešeno přepočítáváním souřadnic tlačítka myši u každé aktivity myši na základě poměru zoomu.

```
protected void drawDiagram(Graphics g) {
        Graphics2D g2 = ((Graphics2D) g);
        g2.scale(zoomX, zoomY);
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        for(GraphicElement element : diagram.getNodes()){
            element.draw(g2);
        }
        for(GraphicElement element: diagram.getConnections()){
            element.draw(g2);
        }
    }
```

Výpis 4: Metoda pro přepočet rozměrů na základě poměru zoomu

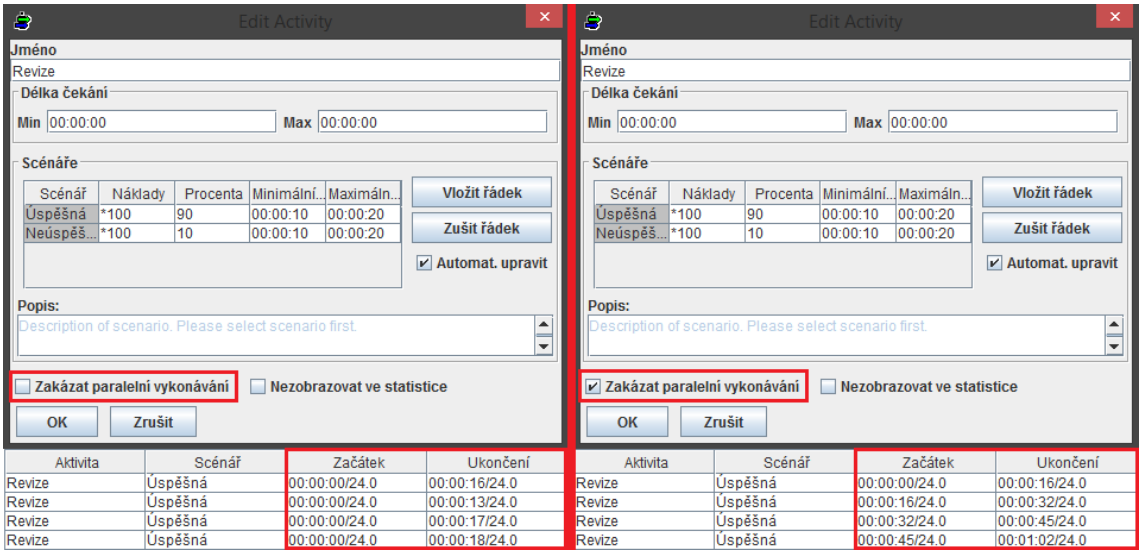
4.3 Zobrazení aktuálního času simulace

Pro monitorování průběhu simulace jsem přidal do okna simulace nový popisek zobrazující aktuální čas simulace. Tento popisek bylo potřeba aktualizovat v pravidelném intervalu, aby zobrazovaná doba byla co nejaktuálnější. Z tohoto důvodu jsem vytvořil

ve třídě AutomaticSimulationRunner, reprezentující okno pro spouštění simulace, nové vlákno, které bylo spouštěno zároveň se simulací. Toto vlákno každou sekundu zaznamenalo aktuální čas simulace uložený v instanci třídy Agenda uvnitř třídy Simulation a tento čas poté vložilo do příslušného popisku.

4.4 Možnost zakázat paralelní vykonávání aktivity

Pro možnost volby, jestli se určitá aktivita může vykonávat paralelně nebo ne, jsem zvolil přidání komponenty JCheckBox v oknu pro definici vlastností aktivity. Tento checkbox nastavoval ve třídě Activity hodnotu proměnné reprezentující maximální počet, kolikrát může daná aktivita současně běžet. Tuto proměnnou bylo poté ve třídě Simulation nutné během simulace kontrolovat a v případě, že bylo zakázané paralelní vykonávání, nedovolit spouštění této aktivity vice než jednou současně.

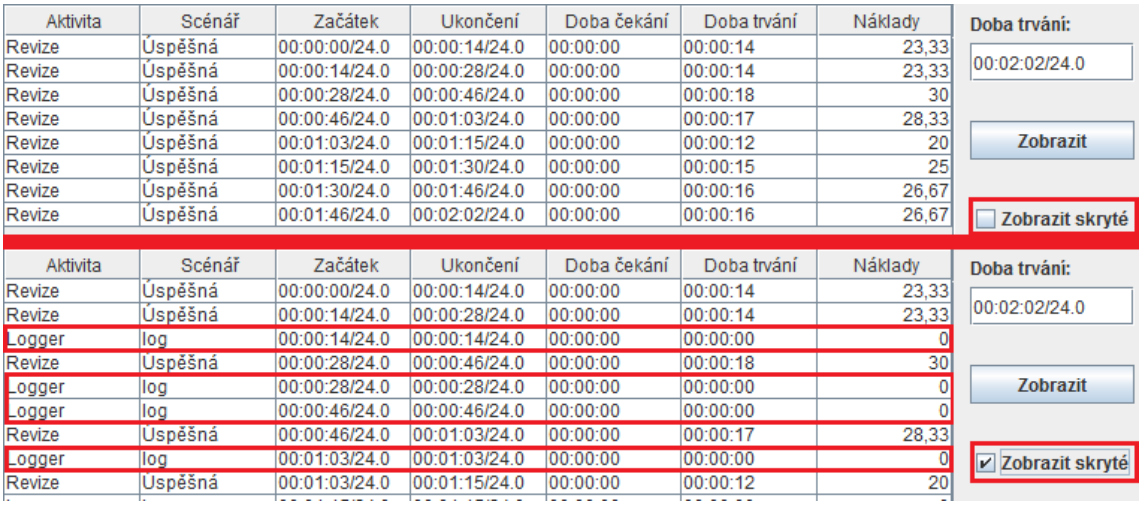


Obrázek 3: Paralelní vykonávání aktivity

4.5 Možnost skrytí aktivit ve výsledcích simulace

Pro nastavení skrytí aktivity ve výsledcích simulace jsem opět využil komponentu JCheckBox, kterou jsem přidal do definice vlastností aktivity. Tento checkbox nastavoval hodnotu proměnné uvnitř třídy Activity reprezentující, zda je aktivita skrytá nebo ne. Při výpisu výsledků simulace do tabulky průběhu aktivit byla následně tato hodnota kontrolována a zvolené aktivity byly vynechány při vypisování záznamů do tabulky. Ovšem občas by mohl uživatel chtít zobrazit i tyto aktivity, které byly nastaveny, aby se nezobrazovaly do výsledků simulace. Určitě by nebylo nejvhodnější, kdyby uživatel musel z tohoto důvodu projít všechny aktivity a zrušit jejich skrytí, aby si je mohl ve výsledcích

prohlédnout. Proto byl přidán i jeden checkbox k tabulce výsledků umožňující zobrazení i skrytých aktivit.



Aktivita	Scénář	Začátek	Ukončení	Doba čekání	Doba trvání	Náklady
Revize	Úspěšná	00:00:00/24.0	00:00:14/24.0	00:00:00	00:00:14	23,33
Revize	Úspěšná	00:00:14/24.0	00:00:28/24.0	00:00:00	00:00:14	23,33
Revize	Úspěšná	00:00:28/24.0	00:00:46/24.0	00:00:00	00:00:18	30
Revize	Úspěšná	00:00:46/24.0	00:01:03/24.0	00:00:00	00:00:17	28,33
Revize	Úspěšná	00:01:03/24.0	00:01:15/24.0	00:00:00	00:00:12	20
Revize	Úspěšná	00:01:15/24.0	00:01:30/24.0	00:00:00	00:00:15	25
Revize	Úspěšná	00:01:30/24.0	00:01:46/24.0	00:00:00	00:00:16	26,67
Revize	Úspěšná	00:01:46/24.0	00:02:02/24.0	00:00:00	00:00:16	26,67

Doba trvání:
00:02:02/24.0
Zobrazit
Zobrazit skryté

Aktivita	Scénář	Začátek	Ukončení	Doba čekání	Doba trvání	Náklady
Revize	Úspěšná	00:00:00/24.0	00:00:14/24.0	00:00:00	00:00:14	23,33
Revize	Úspěšná	00:00:14/24.0	00:00:28/24.0	00:00:00	00:00:14	23,33
Logger	log	00:00:14/24.0	00:00:14/24.0	00:00:00	00:00:00	0
Revize	Úspěšná	00:00:28/24.0	00:00:46/24.0	00:00:00	00:00:18	30
Logger	log	00:00:28/24.0	00:00:28/24.0	00:00:00	00:00:00	0
Logger	log	00:00:46/24.0	00:00:46/24.0	00:00:00	00:00:00	0
Revize	Úspěšná	00:00:46/24.0	00:01:03/24.0	00:00:00	00:00:17	28,33
Logger	log	00:01:03/24.0	00:01:03/24.0	00:00:00	00:00:00	0
Revize	Úspěšná	00:01:03/24.0	00:01:15/24.0	00:00:00	00:00:12	20

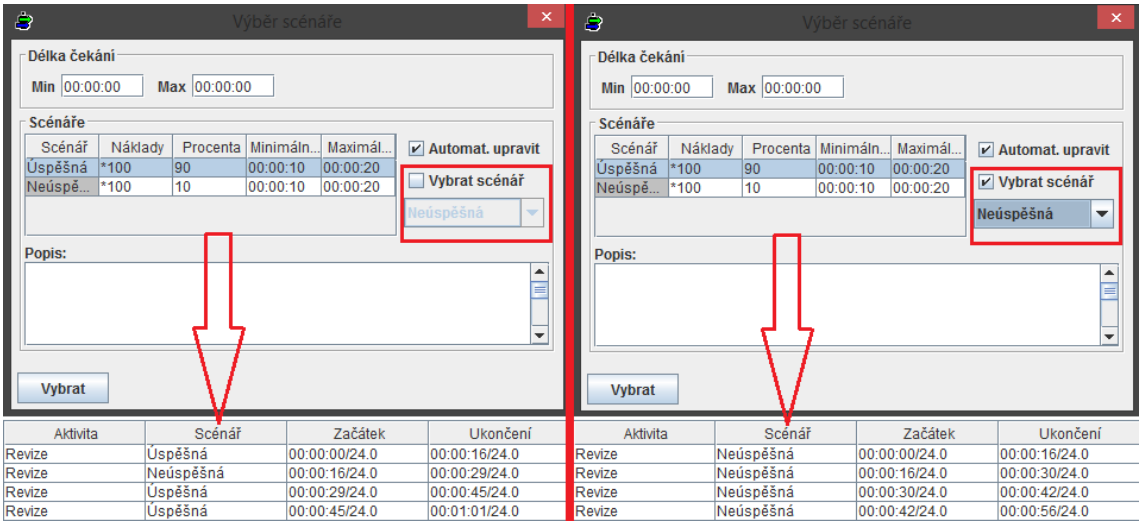
Doba trvání:
00:02:02/24.0
Zobrazit
Zobrazit skryté

Obrázek 4: Aktivity skryté ve výsledcích

4.6 Implementace možnosti volby scénáře

Pro možnost volby scénáře bylo zapotřebí nejprve přidat v simulaci u jednotlivých aktivit prvek umožňující uživatelům volbu, zda se budou vykonávat všechny scénáře náhodně nebo zda se má vykonat jeden určitý scénář. Dále, v případě výběru pouze jednoho scénáře, bylo nutné dodat i prvek pro volbu, který z nich se má vykonat. Nakonec jsem se rozhodl pro implementaci komponenty JCheckBox jakožto rozhodnutí mezi jedním a všemi scénáři a komponenty JComboBox pro určení, který scénář se má vykonat. Tento comboBox jsem naplnil scénáři dané aktivity. V následujícím kroku bylo nutné uložit tuto možnost do dané instance třídy SimulatedActivity, tedy vytvořit novou proměnnou pro uchování stavu checkboxu (zakliknutý/nezakliknutý) a proměnnou pro uchování zvoleného scénáře. Nakonec bylo nezbytné tyto informace využít přímo při simulaci, tzn. ve třídě Simulation u volby scénáře aktivity zjistit hodnotu proměnné uchovávající stav checkboxu a buď nastavit scénář náhodný, nebo načíst scénář z proměnné obsahující informace o zvoleném scénáři.

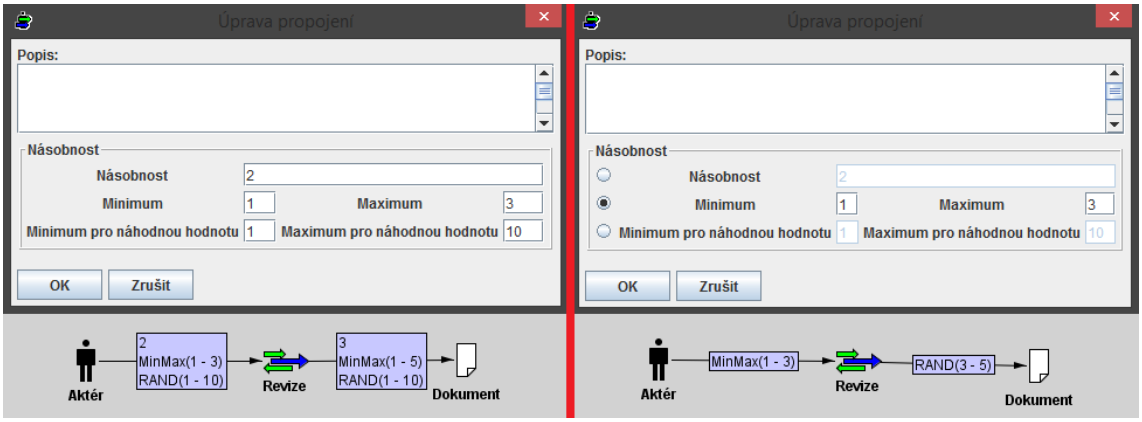
Tímto byla sice volba scénáře plně funkční, nicméně při uložení modelu nebyla uložena i tato volba. To znamená, že při následném načtení modelu byla volba scénáře opět v původním nastavení, což není zrovna nejvhodnější, obzvláště u rozsáhlejších modelů. Nastavení volby scénáře nebylo uloženo z toho důvodu, že hodnoty byly uloženy jen ve třídě SimulatedActivity, která je vytvářena automaticky při simulaci a její stav není při ukládání uložen. Proto bylo nutné informace týkající se volby scénáře uložit i do původní třídy Activity, jejíž atributy jsou již při ukládání uloženy.



Obrázek 5: Možnost volby scénáře

4.7 Úprava vzhledu volby násobnosti

Pro změnu vzhledu volby násobnosti u propojení uzlů jsem využil možností komponenty JRadioButton a ButtonGroup. Uživatel si tak může vybrat jednu ze tří možností násobnosti, která se poté použije při simulaci. Také jsem upravil vzhled zobrazení těchto hodnot v diagramu u jednotlivých propojení a nyní se zobrazují už jen informace týkající se zvoleného typu násobnosti.



Obrázek 6: Změna vzhledu volby násobnosti (vlevo před, vpravo po změně)

4.8 Úpravy a přidávání tabulek s výsledky simulace

S rozšířením možností aplikace bylo nezbytné upravit vzhled některých tabulek a také přidat pár dalších záložek pro zobrazení nových informací a grafů.

4.8.1 Tabulka pro zobrazení časových průběhů aktivit

Jedním z míst, kde byla tato změna potřebná bylo zobrazení grafů časových průběhů aktivit. V původní verzi byl časový průběh zobrazen v instanci typu jTable, kde jeden sloupec obsahoval název aktivity a druhý obsahoval vykreslený časový průběh. Ovšem z důvodu změny i zobrazení těchto grafů jsem místo jTable využil vlastností GridBagLayout, aby zobrazované grafy byly, i bez nutnosti zvětšení grafu, alespoň trochu přehledné. Pro rozložení komponent se využívá třídy GridBagConstraints, která umožňuje nastavení rozložení, chování při změně rozměrů a různých omezení jednotlivých prvků uvnitř GridBagLayout (ukázka kódu 5). Mezi nastavitelné vlastnosti patří např. vlastnost fill určující, zda se komponenta uvnitř buňky rozpíná, aby zaplnila celý prostor buňky, vlastnost gridwidth určující, kolik buněk na řádku zabere daná komponenta, vlastnosti gridx a gridy nastavující, ve kterém řádku a sloupci je komponenta vykreslena, weightx a weighty určující, jak moc se buňka rozpíná horizontálně a vertikálně při změně velikosti tabulky, a další. Po změně vzhledu tabulky je zobrazen název aktivity, zmenšený graf průběhu aktivity a tlačítko umožňující detailní pohled na určitý graf.

```
JPanel grid = new JPanel(new GridBagLayout());
JLabel label = new JLabel("component");
GridBagConstraints gridBagConstraints = new GridBagConstraints();
gridBagConstraints. fill  = GridBagConstraints.BOTH;
gridBagConstraints.weighty = 0.0;
gridBagConstraints.weightx = 1.0;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 0;
grid.add(label, gridBagConstraints);
```

Výpis 5: Ukázka použití GridBagLayout a GridBagConstraints

4.8.2 Tabulka pro zobrazení histogramů aktivit

Dalším z nezbytných kroků bylo vytvoření tabulky umožňující přístup k jednotlivým histogramům aktivit. Zde již nebylo nutné zobrazování zmenšenin grafů, jako tomu bylo u tabulky časových průběhů aktivit, tudíž jsem využil komponenty jTable. Stejně jako tomu bylo při přepisování kódu na využití generických kolekcí u výpisu informací o aktivitách při simulaci, tak i při vytváření histogramu bylo potřeba vytvořit třídu uchovávající data a informace o sloupcích tabulky a třídu reprezentující datový model tabulky. Tabulka pro zobrazení histogramů obsahuje v jednom sloupci názvy aktivit a ve druhém sloupci tlačítko pro zobrazení histogramu. Zatímco u sloupce s názvem aktivity stačilo jako render využít instance třídy DefaultTableCellRenderer z Javax Swing knihovny, pro sloupec s

tlačítkem implementovaným uvnitř buňky bylo nezbytné vytvořit vlastní TableCellRenderer(ukázka kódu 6), jehož instance je poté nastavena jako renderer sloupce zobrazujícího tlačítka (ukázka kódu 7).

```
public class JTableButtonRenderer extends JButton implements TableCellRenderer {

    @Override
    public Component getTableCellRendererComponent(JTable table,
            Object value, boolean isSelected, boolean hasFocus,
            int row, int column) {

        setValue();
        return this;
    }
    public void setValue(){
    }
}
```

Výpis 6: Ukázka vytvoření JTableButtonRenderer třídy pro zobrazení tlačítka v tabulce

```
DefaultTableCellRenderer activityRenderer = new DefaultTableCellRenderer() {
        public void setValue(Object value) {
            if (value instanceof Nameable) {
                setText(((Nameable) value).getName());
            } else {
                setText("");
            }
        }
    };

JTableButtonRenderer buttonRenderer = new JTableButtonRenderer(){
        public void setValue(){
            setText(Public.texts.getString("Zobrazit histogram"));
        }
    };
table.getColumn(Public.texts.getString(" Activity ")).setCellRenderer(activityRenderer);
table.getColumn(Public.texts.getString("Histogram")).setCellRenderer(buttonRenderer);
```

Výpis 7: Ukázka vytvoření instancí rendererů a jejich využití

4.8.3 Tabulka pro zobrazení počtu tokenů v jednotlivých uzlech

Tabulka pro zobrazení počtu tokenů v jednotlivých uzlech zobrazuje při každém kroku simulace seznam všech aktivních a pasivních uzlů vykreslených v každém z diagramů, informaci, zda se jedná o aktivní nebo pasivní uzel, počet tokenů v uzlu a tlačítko pro zobrazení grafu. Pro zobrazení všech těchto informací bylo nejprve nutné vytvořit třídu, která by uchovávala pro daný čas počet tokenů v každém z aktivních a pasivních uzlů (ukázka kódu 8). Pro zobrazení tabulky by stačila jen jedna instance této třídy, ovšem mým cílem nebylo jen zobrazit aktuální počet tokenů v jednotlivých uzlech, ale také následně vytvořit graf znázorňující změny v počtu tokenů v jednotlivých uzlech v průběhu

simulace. K tomu bylo zapotřebí nejen zaznamenat aktuální stavy uzlů, ale také tento stav vložit do kolekce, se kterou se dále pracuje při vytváření grafů. Pro zobrazení tabulky bylo stejně jako u tabulky aktivit a histogramů nezbytné vytvořit třídu uchovávající data zobrazované v tabulce a popis jednotlivých sloupců tabulky a třídu reprezentující datový model tabulky. Také pro zobrazení tlačítka uvnitř tabulky bylo potřeba pro daný sloupec nastavit jako renderer instanci již dříve vytvořené třídy JTableButtonRenderer.

```
public class ActivePassiveLog {
    private Time time;
    private Hashtable<SimulatedNode, Integer> tokensCount;
    public Time getTime() {
        return time;
    }
    public void setTime(Time time) {
        this.time = time;
    }
    public Hashtable<SimulatedNode, Integer> getTokensCount() {
        return tokensCount;
    }
    public void setTokensCount(Hashtable<SimulatedNode, Integer> tokensCount) {
        this.tokensCount = tokensCount;
    }
    public ActivePassiveLog(Time time) {
        super();
        this.time = time;
        tokensCount = new Hashtable<SimulatedNode, Integer>();
    }
}
```

Výpis 8: Ukázka třídy pro uchování počtu tokenů v uzlech v určitém čase simulace

4.9 Knihovny pro tvorbu grafů

Často se může stát, že při programování člověk pracuje s velkým množstvím různých dat, která mohou být na první pohled nepřehledná. Tyto data je možné prezentovat třeba nějakou tabulkou, ale i to je někdy nedostačující. Proto bývá občas požadováno i zpracování dat do formy grafu.

Pro grafickou reprezentaci dat v jazyce Java existuje poměrně dost knihoven, které umožňují vytváření grafů. Patří zde např. knihovny JFreeChart, JCC Kit, Openchart2, GRAL Graphing Library, charts4j, Jzy3D a mnoho dalších.

4.9.1 JFreeChart

JFreeChart je knihovna dostupná zdarma, která usnadňuje vývojářům zobrazení grafů profesionální kvality v jejich aplikacích. Obsahuje konzistentní API s dobrou dokumentací, podporující širokou škálu různých typů grafů. Podporuje mnoho typů výstupu, včetně Swing a JavaFX komponent, obrázkových souborů (PNG, JPEG, ...), PDF, EPS, SVG souborů a dalších. Více informací dostupných na stránkách knihovny JFreeChart[3].

4.9.2 JCC Kit

JCCKit je malá knihovna (< 100Kb) a velmi flexibilní framework pro vytváření grafů. JCCKit je napsán pro platformu JDK 1.1.8, takže je vhodný pro Applety a PDA. Více informací dostupných na stránkách knihovny JCCKit[4].

4.9.3 Openchart2

Knihovna Openchart2 je založena na základě JOpenChart knihovny od Sebastiana Mullera. Nabízí jednoduché rozhraní pro Java programátory pro vytváření 2D grafů. Knihovna nabízí výběr různých stylů grafů, včetně sloupcových a koláčových grafů. Více informací dostupných na stránkách knihovny OpenChart2[5].

4.9.4 GRAL Graphing Library

GRAL Graphing Library je knihovna dostupná zdarma, sloužící pro zobrazování grafů, diagramů atd. Umožňuje vytváření sloupcových, koláčových, spojnicových, plošných, XY bodových, bublinových a dalších grafů. Více informací dostupných na stránkách knihovny GRAL[6].

4.9.5 charts4j

Charts4j umožňuje vývojářům vytvořit grafy dostupné v Google Chart Tools přes Java API. Více informací dostupných na stránkách knihovny Charts4j[7].

4.9.6 Jzy3D

Jzy3D je knihovna poskytující možnost vykreslení 2D i 3D grafů. API poskytuje podporu pro interaktivní grafy, popisky, libovolné nastavení os a nákresu. Více informací dostupných na stránkách knihovny Jzy3D[8].

4.10 Implementace grafů

Pro vytváření grafů v mé bakalářské práci mi byla vedoucím práce doporučena knihovna JFreeChart. Výhodou této knihovny je mimo jiné i přístupnost k libovolné části kódu a možnost modifikovat, rozšiřovat či jinak manipulovat s kódem dle vlastní potřeby a požadavků. Dále taky nabízí možnost tvorby jak 2D, tak i 3D grafů různých typů, ať už to je koláčový graf, sloupcový graf, skládaný sloupcový graf, plošný graf, čárový graf, Ganttův diagram, histogram nebo jiné, kde pro každý typ grafu je dostupných hned několik různých ukázkových příkladů, díky kterým si vývojář může prohlédnout a osvojit způsob vytváření daného typu grafu.

Tvorba grafu by se dala rozdělit na několik částí. Nejprve je nutné vytvořit dataset, tedy množinu dat, ze kterých je graf vykreslen. Dále se vytvoří instance typu JFreeChart, k čemuž lze využít statickou třídu ChartFactory. Ta poskytuje velké množství metod

pro vytváření různých grafů, kde jako parametry metod se předávají informace o názvu grafu, popisku os, dříve vytvořený dataset, orientace grafu a informace, zda má být zobrazena i legenda, případně popisky. Poté si můžeme z instance JFreeChart vytáhnout instanci třídy Plot, díky které lze nastavit libovolný vzhled grafu, ať už to je barva pozadí, vzhled zaznamenaných datových množin, vzhled nebo rozsahy os atd. Posledním krokem je vytvoření panelu, na který se graf bude zobrazovat.

4.10.1 Časový průběh simulace

Prvním z požadovaných grafů, které jsem měl zobrazit, byl graf znázorňující časový průběh simulace a počet paralelně běžících aktivit v dané chvíli. K tomu jsem využil vykreslení pomocí čárového grafu. První a zároveň nejsložitější částí tvorby grafu bylo vytváření datasetu, tedy vytvoření množiny dat, která by zobrazovala změny v simulaci. K tomu jsem využil seznam Recordů, které uchovávají informace o čase začátku aktivity, době trvání aktivity a dobu ukončení aktivity. Tyto data jsem potřeboval pro vytvoření grafu, a proto jsem seznam Recordů prošel a pro patřičný element uložil data do kolekcí.

```
private XYDataset createDataset(ArrayList<Record> data, ProcessElement element){
    ArrayList<Long> starts = new ArrayList<Long>();
    ArrayList<Long> ends = new ArrayList<Long>();
    Time finishLast = new Time(0);
    for(Record r : data){
        if (r.getMaxTime().getTime()>finishLast.getTime())
            finishLast = r.getMaxTime();
        if (r. getActivity () .getElement() == element)
        {
            if (r.getRunStart() != null){
                starts.add(r.getRunStart().getTime());
                ends.add(r.getRunStart().getTime()+r.getRunDuration().getTime());
            }
        }
    }
```

Výpis 9: Získání seznamů dob začátků a ukončení aktivit

Dále, protože v daný časový okamžik mohlo začít, případně skončit i několik paralelně vykonávaných procesů, jsem tyto data uložil do kolekce Hashtable zaznamenávající pro každý okamžik změny v počtu běžících procesů.

```
Hashtable<Long, ArrayList<Long>> graphPoints = new Hashtable<Long, ArrayList<Long>>();
Long countRunning = 0L;
int endsIndex = 0;
int startsIndex = 0;
for(int i=0; i < (starts.size()+ends.size()); i++){
    if (startsIndex>=starts.size()){
        if (graphPoints.containsKey(ends.get(endsIndex))){
            graphPoints.get(ends.get(endsIndex)).add(--countRunning);
            endsIndex++;
        }
        else{
            ArrayList<Long> newOne = new ArrayList<Long>();
```

```
                newOne.add(countRunning);
                newOne.add(--countRunning);
                graphPoints.put(ends.get(endsIndex), newOne);
                endsIndex++;
            }
        }
        else if (ends.get(endsIndex)<=starts.get(startsIndex) && countRunning>0){
            if (graphPoints.containsKey(ends.get(endsIndex))){
                graphPoints.get(ends.get(endsIndex)).add(--countRunning);
                endsIndex++;
            }
            else{
                ArrayList<Long> newOne = new ArrayList<Long>();
                newOne.add(countRunning);
                newOne.add(--countRunning);
                graphPoints.put(ends.get(endsIndex), newOne);
                endsIndex++;
            }
        }
        else{
            if (graphPoints.containsKey(starts.get(startsIndex))){
                graphPoints.get(starts.get(startsIndex)).add(++countRunning);
                startsIndex++;
            }
            else{
                ArrayList<Long> newOne = new ArrayList<Long>();
                newOne.add(countRunning);
                newOne.add(++countRunning);
                graphPoints.put(starts.get(startsIndex), newOne);
                startsIndex++;
            }
        }
    }
```

Výpis 10: Získání hodnot pro vykreslení grafu

Nakonec jsem tuto kolekci prošel a všechny hodnoty zaznamenal do Série, která se vložila do datasetu.

```
    XYSeries localXYSeries = new XYSeries("");
    localXYSeries.add(new Time(0).getTime(), 0);
    for(Long key : Collections.list(graphPoints.keys())){
        ArrayList<Long> values = graphPoints.get(key);
        for(Long value : values){
            localXYSeries.add(key, value);
        }
    }
    localXYSeries.add(finishLast.getTime(), 0);
    XYSeriesCollection localXYSeriesCollection = new XYSeriesCollection();
    localXYSeriesCollection.addSeries(localXYSeries);
    return localXYSeriesCollection;
}
```

Výpis 11: Vytvoření a naplnění datasetu

Poté už šlo jen o vytvoření instance typu JFreeChart, nastavení vzhledu grafu a jeho zobrazení na panelu.

4.10.2 Histogram znázorňující intervaly doby vykonávání procesu

Dalším z požadavků mé práce bylo vykreslení histogramu znázorňujícího rozčlenění doby vykonávání aktivit do několika časových intervalů a zobrazení počtu, kolikrát se daný proces vykonal v určitém intervalu v průběhu jedné, či více simulací spouštěných opakovaně. K tomu bylo zapotřebí vytvořit si ve třídě Simulation kolekci, která by uchovávala historii všech najednou spuštěných simulací.

Při vytváření histogramu bylo nejdříve zapotřebí projít všechny zaznamenané historie a získat hodnoty minimálního a maximálního času, po který se může proces vykonávat a také doby trvání jednotlivých procesů.

```
private IntervalXYDataset createDataset(ArrayList<History> data, ProcessElement element){
    Time minDurationTime = null;
    Time maxDurationTime = new Time(0);
    ArrayList<Long> durations=new ArrayList<Long>();
    for(History history : data){
        for(Record r : history.getRecords()){
            if(r.getActivity().getElement() == element)
            {
                durations.add(r.getRunDuration().getTime());
                Time minDur = new Time(r.getScenario().getMinDuration());
                Time maxDur = new Time(r.getScenario().getMaxDuration());
                if(maxDur.getTime()>maxDurationTime.getTime()){
                    maxDurationTime = maxDur;
                }
                if(minDurationTime == null){
                    minDurationTime = minDur;
                }
                else if(minDur.getTime()<minDurationTime.getTime()){
                    minDurationTime = minDur;
                }
            }
        }
    }
```

Výpis 12: Získání minimální a maximální doby trvání aktivity

Nyní, když už jsem získal interval určený pro vykonání procesu, bylo potřeba tento interval rozdělit na několik dílčích intervalů.

```
ArrayList<Long> bounds = new ArrayList<Long>(6);
Long duration = maxDurationTime.getTime()−minDurationTime.getTime();
if(duration>=5){
    for(int i=0; i<=5;i++){
        bounds.add(((maxDurationTime.getTime()−minDurationTime.getTime())/5)∗i+
            minDurationTime.getTime());
    }
}
else if(duration >= 3){
```

```
    for(int i=0; i<=3;i++){
        bounds.add(((maxDurationTime.getTime()−minDurationTime.getTime())/3)∗i+
            minDurationTime.getTime());
    }
}
else if(duration >=1){
    bounds.add(minDurationTime.getTime());
    bounds.add(maxDurationTime.getTime());
}
```

Výpis 13: Získání hraničních hodnot intervalů

Dále, když už jsem měl vytyčeny jednotlivé intervaly, musel jsem vytvořit bloky pro dané rozmezí a do nich zařadit doby trvání procesů.

```
ArrayList<SimpleHistogramBin> bins = new ArrayList<SimpleHistogramBin>();
for(int i=0; i<(bounds.size()−1);i++){
    if(i == (bounds.size()−2))
        bins.add(new SimpleHistogramBin(bounds.get(i), bounds.get(i+1), true, true));
    else
        bins.add(new SimpleHistogramBin(bounds.get(i), bounds.get(i+1), true, false));
}
ArrayList<Integer> counts = new ArrayList<Integer>(bounds.size()−1);
for(int i=0;i<bounds.size()−1;i++)
    counts.add(0);
for(Long d : durations){
    if((d>=bounds.get(0) && d<bounds.get(1)) || (d>=bounds.get(0) && d<=bounds.get(1) &&
        bounds.size()==2)){
        counts.set(0, counts.get(0)+1);
    }
    else if(bounds.size()>=4){
        if(d>=bounds.get(1) && d<bounds.get(2)){
            counts.set(1, counts.get(1)+1);
        }
        else if((d>=bounds.get(2) && d<bounds.get(3)) ||
            (d>=bounds.get(2) && d<=bounds.get(3) && bounds.size()==4)){
                counts.set(2, counts.get(2)+1);
        }
        if(bounds.size()>=6){
            if(d>=bounds.get(3) && d<bounds.get(4)){
                counts.set(3, counts.get(3)+1);
            }
            else if(d>=bounds.get(4) && d<=bounds.get(5)){
                counts.set(4, counts.get(4)+1);
            }
        }
    }
}
```

Výpis 14: Vytvoření a naplnění bloků histogramu

Poté už stačilo jen bloky vložit do datasetu, ze kterého se vykreslí graf.

```
    SimpleHistogramDataset histogramDataset = new SimpleHistogramDataset("");
    for(int i=0; i< counts.size(); i++){
        bins.get(i).setItemCount(counts.get(i));
        histogramDataset.addBin(bins.get(i));
    }
    histogramDataset.setAdjustForBinSize(false);
    return histogramDataset;
}
```

Výpis 15: Naplnění datasetu

Nakonec už opět šlo jen o vytvoření instance typu JFreeChart z vytvořeného datasetu, nastavení vzhledu histogramu a jeho zobrazení na panelu.

4.10.3 Graf zobrazující pohyb tokenů v uzlu

Poslední z informací, které bylo třeba zaznamenat grafem, byl počet tokenů v aktivních a pasivních uzlech v jednotlivých krocích simulace. K tomu bylo využito kolekce, kterou jsem vytvořil při zobrazení tabulky aktuálních počtů tokenů v uzlech. Pro zobrazení tohoto grafu jsem opět vybral čárový graf, kde při vytváření datasetu se nejprve vyhledal simulovaný uzel a poté se prošla kolekce záznamů v jednotlivých časech simulace a do série se zaznamenala hodnota aktuálního počtu uzlů v grafu pro tento čas.

```
private static XYDataset createDataset(ArrayList<ActivePassiveLog> data, ProcessElement el) {
    SimulatedNode simulated = null;
    for(SimulatedNode node : Collections.list(data.get(0).getTokensCount().keys())){
        if(node.getElement() == el){
            simulated = node;
            break;
        }
    }
    XYSeries localXYSeries = new XYSeries("");
    if(simulated != null){
        ActivePassiveLog previous = null;
        for(ActivePassiveLog log : data){
            if(log.getTokensCount().containsKey(simulated)){
                if(previous != null)
                    localXYSeries.add(log.getTime().getTime(), previous.getTokensCount().get(
                        simulated));
                localXYSeries.add(log.getTime().getTime(), log.getTokensCount().get(simulated));
                previous = log;
            }
        }
    }
    XYSeriesCollection localXYSeriesCollection = new XYSeriesCollection();
    localXYSeriesCollection.addSeries(localXYSeries);
    return localXYSeriesCollection;
}
```

Výpis 16: Metoda pro vytváření datasetu u grafu zobrazujícího pohyb tokenů v uzlu

5 Závěr

Cílem této bakalářské práce bylo upravení a vylepšení možností programu BP Studio a seznámení s tím, co vše obnáší takováhle úprava programu. Hned prvním bodem práce bylo přepsání kódu na využití generických kolekcí, což bylo asi nejzdlouhavější už z toho důvodu, že bylo potřeba přepsat spousty úseků kódu, případně vyřešit problémy týkající se práce s tímto kódem. Zpočátku bylo pro mě velice náročné zorientovat se v kódu a v tom, co má která třída na starost. Často jsem se kvůli využití netypických vektorů v kódu zamotal a zabralo mi spoustu času, než jsem daný kód pochopil. Postupem času, při přepisování kódu, vektorů ubývalo a pro mě se zdál být obsah tříd více přehledný. I přes veškerý čas, který mi zabral pouze tento bod bakalářské práce, jsem si po jeho dokončení uvědomil, že to byl asi jeden z nejvhodnějších způsobů, jak si zvyknout na tento styl psaní kódu a také byl vhodný pro získání přehledu o tom, k čemu která třída slouží a jak jsou vzájemně propletené. Díky tomu již bylo mnohem jednodušší implementovat ostatní změny v programu. I přes překážky, na které jsem narazil v průběhu práce, se mi nakonec podařilo implementovat všechny změny a přidat všechny požadované funkcionality do aplikace.

Díky této práci jsem si uvědomil, že i když jde pouze o úpravy existujícího programu, můžou i ty být poměrně náročné, obzvláště pro někoho, kdo se nepodílel na vývoji původní verze programu. Touto prací jsem si také vylepšil znalosti programování v jazyce Java a naučil jsem se i pracovat se swing komponentami. Další přínosnou věcí bylo i získání přehledu o dostupných knihovnách pro vytváření grafů v jazyce Java a v důsledku implementace grafů do této bakalářské práce i získání znalostí, jak se pracuje s grafy při využití knihovny JFreeChart.

Horáček Ondřej

6 Reference

[1] Business Process Studio, *About the BP Studio* [online], Ivo Vondrák, 25.9.2000, [cit. 30.4.2015], dostupné z : http://vondrak.cs.vsb.cz/download/bpstudio_eval/manual.pdf

[2] wikipedia, *Discrete event simulation* [online], [cit. 30.4.2015], dostupné z : http://en.wikipedia.org/wiki/Discrete_event_simulation

[3] JFree, *JFreeChart* [online], dostupné z: http://www.jfree.org/jfreechart/ k 3.5.2015

[4] jcckit.sourceforge.net, *Chart Construction Kit* [online], dostupné z: http://jcckit.sourceforge.net/ k 3.5.2015

[5] Approximatrix, *Openchart2* [online], dostupné z: http://approximatrix.com/products/openchart2/ k 3.5.2015

[6] trac.erichseifert.de, *Gral Java Graphing* [online], dostupné z: http://trac.erichseifert.de/gral/ k 3.5.2015

[7] code.google.com, *charts4j* [online], dostupné z: https://code.google.com/p/charts4j/ k 3.5.2015

[8] jzy3d, *JZY3D* [online], dostupné z: http://jzy3d.org/ k 3.5.2015

A CD s aplikací

Přiložené CD obsahuje:

- Text této bakalářské práce ve formátu PDF
- Zdrojové kódy programu
- Spustitelný soubor (BPModel.jar)